CMS Calibration, Alignment, and Databases Roadmap

Contributors L. Lueking, F. Glege, V. Innocente, O. Buchmuller, Y. Guo, G. Lukhanin, S. Kosyakov, M. Case, C. Jones, G. Bruno, Z. Xie, S. Iqbal, and N.E. Others,

(Date:May 13, 2005)
Revision 1.1

Contents

List of Figures

1 Introduction

1.1 Purpose of this Document

The CMS Calibration, Alignment, and Database (CalAliDB) Project is charged with understanding the database needs and coordinating their design, implementation, and deployment. This road map contains information pertaining to the nature of the various detector sub-systems in CMS, and use cases for how the databases will be used, and discussion of how they will be built. Included in this road map are suggestions for design tools, rules, and a procedure for review. The road map includes both online and offline database needs, and the relationships between them. Several specific logical areas have been identified by CMS as focus points for database design, the distinction between these various types of data is will become better understood as the project progresses and the required relationships will be explored.

This document is a roadmap to aid the development of the Calibration, Alignment and Database software and operation. It contains many sections that are incomplete, and will continue to do so for the foreseeable future. It contains discussions of requirements, system architectures, design guidelines, details of the design, and plans for the future.

1.2 Structure of this Document

No section of this document is “final”. All are subject to change.

Sections marked . . .

Commentary from: John Smith

. . . like this

are notes added by one (or more) authors, but which have not yet been “approved” as an official part of the document. Section marked *like this* are specially called out as incomplete sections.

Sections marked with like this:
"**Approved on 7 Oct 1571**"
have been officially approved. This means that we consider the points in such a section settled. In order to re-open such a point for discussion, one needs to make a persuasive argument that the related analysis is incorrect or incomplete, and to persuade the others that some new analysis is better.

In many parts of this document, we refer to examples of code, configuration information, and other information. While we strive to make the examples realistic, we note that they may be for expository purposes only.

1.3 Scope of the Project

The project includes two major sets of deliverables:

1. Database schema designs for online and offline applications related to CalAli areas. The solutions used for online and offline are possibly different based on the requirements for each environment.
2. A consistent set of API's compatible with the EDM framework (for offline), for accessing the data managed in the schemas. Online API's for those areas not covered by existing online work.
3. Deployment and long term maintenance strategy for the developed applications and databases.

2 Requirements and Use Cases

2.1 Overview

In an effort to understand the database needs of CMS, four logical areas have been identified as relevant to the discussion 1) Construction , 2) Equipment management 3) Configuration, and 4) Conditions (for a more detailsed description refer to [1]). The construction database tracks the process of building each subsystem, and includes details of how parts are brought together and various levels of component testing. Equipment management stores information for the smallest replaceable component, and was intended originally to monitor each components level of radiation exposure to the beam, and keep an inventory of all parts added to the detector. The Configuration database stores parameters that are loaded into the front-end electronics of each sub-detector to operate it. Finally, conditions include calibration, alignment, and monitoring data that describe the "condition" of all systems when data was being taken. The divisions between these areas are not always clear and there are many relationships needed among them.

There are many subsystems for which databases will be used, and there is a large overlap in the types of data stored and the manner in which it is used. There are eight sub-detectors included in the picture: 1) Pixel, 2) Tracker, 3) ECAL, 4) HCAL, 5) CSC, 6) RPC Barrel and RPC Endcap, 7) Muon DT, and 8) Preshower. In addition, there are additional subsystems that will need databases including Global Trigger, and DAQ. Data is collected and used both on and offline, and the requirements can be quite different for these two cases. In spite of the broad range of needs, there are many areas of commonality and it was felt that a common design was possible to cover many of the needs. An effort to provide a set of common solutions will greatly reduce the effort needed to build and maintain the software and tools in the long run.

2.2 Conditions

Each sub-system will have data about the detector, its response, and operation, which needs to be stored and referenced later. There are several kinds of conditions data: 1) online sensor monitoring, 2) calibration, and 3) alignment data. In the first case, the sensor data originates from monitoring temperatures, currents, voltages, and other ”conditions” at thousands of strategic points on the detector. This data has the following characteristics: 1) each channel is independent, i.e. it can vary in time relative to the other channels, and 2) the channels have a different geometry scheme or layout than the electronic signal channels used to capture the physics data from the detector. The calibration data is generally the measured electronic response signals for each channel of the physics detector, based on a known input condition. The input condition can be charge injection, laser light, radio active source, or actual particle interactions in the detector. Alignment data is needed to precisely know the location of each detector element for any time during the commissioning or running period. The alignment is understood to be a vector translation relative to a known geometry description for each part of the detector.

Each of these types of conditions data will be characterized by a concept known as Interval of Validity (IOV). The IOV is a region in time for which the values of monitoring, calibration, or alignment are valid. An additional feature related to IOV management that is required is the ability to “tag” a set of detector related IOV’s for easy identification. Both IOV, and the ability to have tags are needed to provide the ability to establish a consistent set of metadata when the physics data is processed and analyzed offline.

2.2.0.1 Monitoring

Monitoring is characterized by the fact that each channel can vary independently of other channels, and the IOV for a given channel is the time interval for which the measured sensor value does not change. PVSS helps to manage this with value ranges for each channel. While the sensor measurement is within a particular value range, PVSS will not report a value change to the conditions database. Monitoring data only has one version, that which is taken initially.

The two modes of accessing the data are typically 1) gathering the values for many sensors for a particular time, and 2) collecting the values of one sensor over a range of times. These two access modes would require different schema designs were full optimization desired for each.

2.2.0.2 Calibration

Calibration is based on detector signal measurements taken under controlled conditions. This data is characterized by several parameters per electronic channel, e.g. pedestal, gain, time offset, drift velocity, and others. The data is often collected for sets of channels within a given sub-detector and an algorithm is used to produce a calibration set that is considered valid for a certain interval of time called Interval Of Validity (IOV). This IOV has a beginning time (since time) but is not constrained by an end time (till time) until a new set of calibration data is provided. The calibration information for any given IOV can have one or more versions of the calculation of the calibration values made with different algorithms. This version information might be made of both an algorithm name and version to allow for different algorithms, and slight changes, bug fix, etc., to a particular algorithm. Alignment data is similar to calibration, but has a completely different origin, generally longer IOV periods, and the relative alignment with respect to other detectors is important.

The procedure varies considerably from detector to detector. HCAL uses a radioactive source fed through a small tube to each tower as the starting point of its gain calibration. The signals produced by this source are recorded as a function of the position of the source in the module, and a gain calculated based on an algorithm. Pedestals are measured when the module is completely ”quiet”, with no sources of energy. Electronic noise causes the pedestals to fluctuate and an algorithm is used to determine the best value for each channel. It is anticipated that the HCAL gain and pedestals will not vary significantly over many days or weeks of operation. The ECAL calibration is much more challenging due to the changing response of the lead-tungstate crystals with radiation exposure, and the sensitivity of the gains of the APD to current and temperature variations. A laser pulse is injected into each crystal approximately every 20 minutes and the resulting output signal is used to establish the gain. Figure 1 on the facing page

The actual mapping of the calibration and IOV for HCAL is illustrated by Figure 1 below. Calibration data is taken for particular channels at varying times. This information is processed with a given algorithm into sets of channel calibrations. As shown in the figure, the sets from various calibration runs may be overlapping. Therefore, a mapping procedure is needed to define which calibrations for which channels should be used. In the lower part of the figure, the calibration as it is mapped is shown for two times, t1 and t2.

(needs more detailed description)

2.2.0.3 Alignment

As far as the general specifications for database development are concerned , alignment can be considered as a special type of calibration and, therefore, also exhibit a well defined interval of validity (IOV). Alignment correction usually refer to a set of 6 parameters that represent three translation and three rotation corrections which are defined wrt to the ideal geometrical position o f an active detector element (e.g.silicon strip sensor for the tracker). These corrections need to be applied in order to transform the initially as-

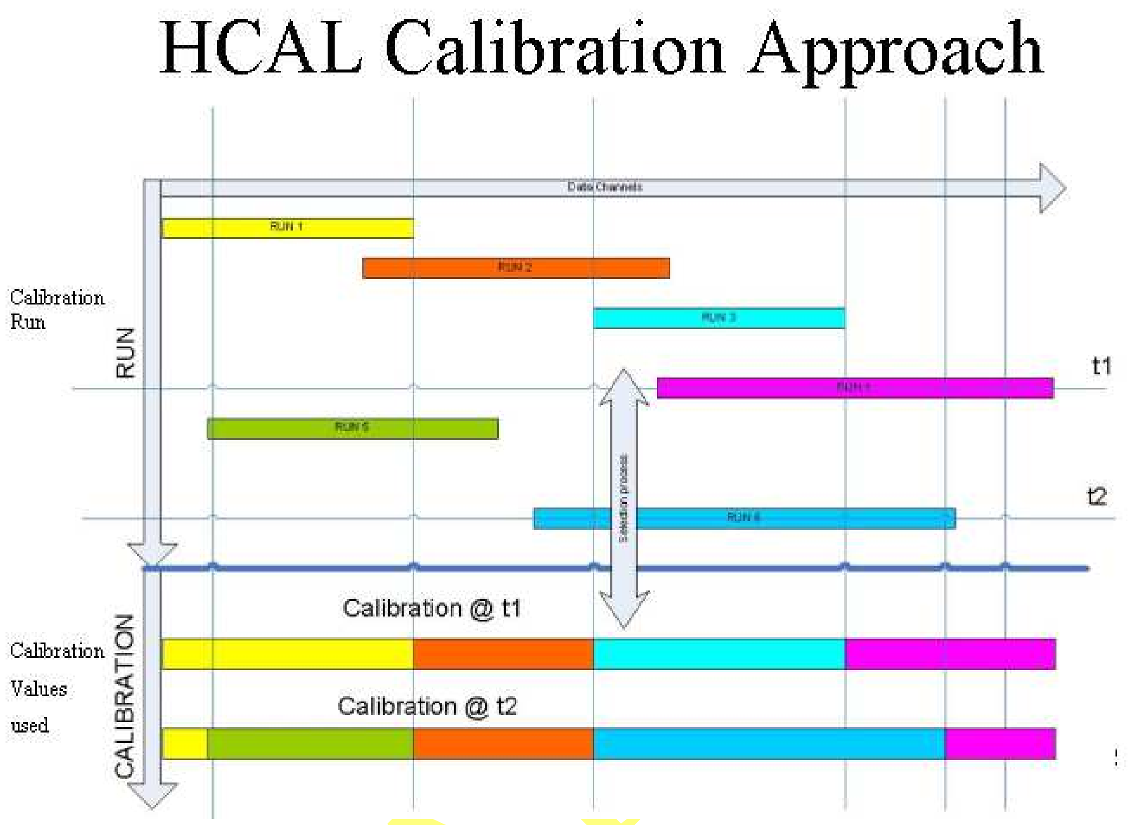

Figure 1: Calibration management.

sumed ideal detector geometry into a realistic one that can be utilized within the various software frameworks.

The most stringent demands on the condition database service regarding alignment are determined by the alignment requirements of the three major tracking devices in CMS (PIXEL, Tracker and Muon Chambers). Especially the hardware alignment motoring systems of Tracker and Muon chambers can provide alignment corrections for larg support structures every couple of minutes during normal data taking. Such a frequent monitoring is important to insure that track based High Level Trigger (HLT) algorithms operate with up-to-date geometry information. Therefore, it is necessary that the alignment condition service not only operates at genuine offline level but also exhibits the possibility to function at HLT.

2.2.0.4 IOV and IOV tagging

The monitoring, calibration, and alignment data is stored in the database, and accessed based on its Interfal of Validity. The IOV is the range in time a value, or set of values is consistent with the running conditions for the detector. Since there are many IOVs spanning a long range of running for all of the relevant values, it is important to have a way of tagging them, similar to the tagging used for CVS code. In addition, the ability to have recursive tagging is needed so a chain, or tree of related tags can be referred to easily. As an example, one might have a “ProductionReconstruction” tag that references all the tags for each subdetector over the entire period of CMS running. By referring to this single tag, one is assured that a consistent set of calibration and alignment is being used. To maintain these tags, responsible persons from each (needs additional specification)

2.3 Several Typical Use Cases

2.4 Private and Community Database instances

Both local (private) and common/central (community) instances of database are needed for the many phases of software and data development. It is important that the interface (API) stays the same independent of the source of the data! Following is a list of use cases for which a local instance of the conditions DB would be useful:

- DB developer:

  A developer/designer of database schema (normalized, and not) and its client code will need many iterations on the schema, then populate a DB, test and optimize code. Most probably the easest way is to scrap the whole DB each time. The developer will reside in her/his own institute and may work on a laptop. The best DB technology for this activity may be SQLight or something similar.

- Calibration algorithm developer:

  Those developing algorithms to create calibration data will need access to a limited sample of event-data and ’old’ conditions DB data. They will develop and optimize code to produce new calibration/alligment constants and will work in a his/her own institure and may work on a disconnected laptop. They will need to populate a DB with the new calibration/alligment constant to test that indeed improve the results running the reconstruction program on a control sample of event-data Data requirements are:

  1. Calibration event-stream
  2. Sub-sample of cond/cal DB corresponding to the event-data
  3. Local DB to be populated (several times) with new calib constant
  4. Control sample of event-data to test the new calib constant Here, the ability to merge seamlessy a read-only db with a small new db is important (eventually creating a new IOV ”index”) A local DB may be helpful and actually unavoidable.

- Reconstrution algorithm developer:

  Similar to the previous item above, but no need to populate a new DB. What is needed is only a read-only local cache of the cond/calib/allign data used in the algorithm under development. It may need an in depth final test on a large dataset. It is not excluded that a ”very-persistent” froNtier squid-cache can do this efficiently and effectively.

- Production/test of new Calibration:

  Although this is a production activity, it is expected that new calib/allign constants must be validated running reconstruction against a control sample of event-data before loading the new constants in the global DB. An environment similar to the one of calib-algo development (with a small local DB containing the new data) may therefore be useful to avoid polluting the global DB with wrong data

- Analysis:

  Analysis will require access to some condition data such as Lumi, detector-status (bad run list?), or even average-beam position. This information may be cached in the event-file, but new improved values may be available at later time. Therefore being able to extract this info into a small local DB seems to be the best solution.

3 Architecture and Technology

Oracle will be used for all production database services and storage of non-event data, both online and offline. There will be one online, one HLT, and one offline instances of databases (See Deployment Chapter 14 on page 29). Distributed access to read-only database information for offline and HLT will be through web-based proxy and caching services (Chapter 12 on page 28). Write access to the offline database will be authenticated through Grid authication mechanisms used by LCG (presumably?).

4 Database Environments and Procedures

(Saima Iqbal, Lee, CERN and FNAL DBA - names TBD soon)DB environments and procedures. (CVS, DB instances, development, testing, production).

This document is developed in order to make sure the controlled development of the databases and their respective API’s (Application Program Interface for the storage and access of data from the respective databases), according to the database requirements of each CMS sub-detector. This proposal/plan is in developing stage and presents a configuration management proposal/plan for CMS-Database project’s Road Map. In this document following configuration management questions are addressed: who will be responsible for and have authority over configuration management; standards, procedures, and guidelines for the project team to follow; tools, resources, and facilities to be used for configuration management; how does developers and project team members request and retrieve configuration control items and what will be the location of controlled items.

4.1 Configuration Management Organization

CMS-Database project manager will decide about team/member/person who will be the responsible for the configuration management of this project. On one side this project is providing support to sub-detectors to develop the databases required in order to support the respective subdetector operation. On the other hand it is integrating and trying to bring the existing subdetectors databases and their respective API's according to the CMS online and offline database requirements as whole and respective standards. It means that sub-detectors who are already very advanced in their database developments, e.g. ECAL and HCAL, already implementing some configuration management rules for their database development. Thus, in order to propose the configuration management and responsible for this management need to study/take a look/consult/integrate these configuration management rules as well.

4.2 Configuration Items

Following software components and documents are required to configure for the CMS-Database project: project road map and other related documents like subdetectors requirement specification documents; database instances; API's to access and store data in the respective online and offline databases. The software components needed to configure for this project is mainly divided in to two major groups: development of each subdetector's database and development of API's for the storage and access of data from the developed databases.

4.2.1 Database Configuration Items

Following suggestion/proposal are made for the configuration of database instances. These suggestions are made on the basis of the suggestion/proposal to use relational databases (example: Oracle) for the storage of non-event data:

Each subdetector is assigned space (data files in terms of the Oracle database technology) on the database server for each of the following proposed software engineering process i.e. development, integration and production of the database instances. [This point is needed to discuss in detail further that whether CMS-DB project manager wants to develop separate database schema for each subdetector or want to create data files with unique name in a single database schema for each subdetector. In addition the database technology/technologies wanted to use in order to clearly define the standards for the configuration of database items]. In the context of this project following are the definitions of the development, integration and production database instances (these definitions are taken from Dirk Duellmann presentation "database/system support for CMS database servers" given at Fermi Lab):

- PRODUCTION - A production database will be used for fully working database applications and will house only ”real” data.
- INTEGRATION - An integration database will be used for testing pre-production database applications. Occasionally, the integration instance may be refreshed with production data.
- DEVELOPMENT - A development database will be used for database and application development, testing of database features.

Following is the syntax to uniquely assign the schema owner name for each subdetector database schema under a single database instance (needs to be reviewed in context of LCG3D counter proposal.See: http://agenda.cern.ch/fullAgenda.php?ida=a052188):

```
CMS_[sub-detector]_[operational level]
```

The ”operational level” is DEVELOPMENT, INTEGRATION, or PRODUCTION, and is included as a safety precaution to avoid any confusion when the developer or DBA is working on a schema. For Example:

```
CMS_PIXEL_DEV[INT,PRD] (schema owner for PIXEL)
CMS_TRACKER_DEV[INT,PRD] (schema owner for Tracker)
CMS_ECAL_DEV[INT,PRD]
CMS_HCAL_DEV[INT,PRD]
CMS_CSC_DEV[INT,PRD]
CMS_RPC_DEV[INT,PRD]
CMS_MUONDT_DEV[INT,PRD]
CMS_PRESHOWER_DEV[INT,PRD]
```

To store the development, integration and production instances of the respective database schemas; under the assigned space for each subdetector, separate portions of spaces are needed to create (in Oracle these portions are termed as Table Spaces) Mainly following are the three types of table spaces which will create for each schema: DATA, INDEX, and BLOBS. Following syntax shows the naming convention for table spaces:

```
CMS-[sub-detector]_[online or offline]_[DATA or INDEX or BLOBS or other]
For example:
CMS_PIXEL_DATA (PIXEL DB DATA)
CMS_PIXEL_INDEX (PIXEL DB indexes)
CMS_PIXEL_BLOB (PIXEL DB blobs)
```

4.2.2 Database API Configuration Items

Database API configuration items. Need to discuss

4.3 Project related Documents

Following documents are needed to configure (store updated/changed versions of the documents) for this project: Project Road Map; Sub-detectors requirements specification document; sub-detectors software testing document [please add the name of the documents expected to develop for this project]

4.4 Versioning of documents and different Software Components

Need group discussion

4.5 Repository for the Configuration Items

CVS is decided to use for the controlled repository of the configuration items. CMS-Database team is geographically distributed thus it is very important to have this CVS repository at easily accessible location. In the initial stage of this project it is proposed to use the CVS setup at Fermi Lab and at later stages of development for the repository of code the CVS facilities available at CERN will be use.

Proposed CVS hierarchal structure for the controlled repository of the database instances, database API and documents (needs to be reviewed in context of CMS CVS repository reorganization and current understanding of CalAliDB needs):

```
Calalign          (domain for calibration and alignment general stuff)
        |
        +- CalalignDB (general source sode) ( at this level each has /doc, /src, /i
                 |
                 + src/    (source code)
                         |
                         + cpp/ C++ code
                         + java/ Java code
                         + python/ Python code

        +- CalalignDML  (Initial Data)
        +- CalalignMigrate (Schema migration scripts)
        +- CalalignScripts (data loading and maintenance scripts)
        +- CalalignDDL  (ddl scripts)
        +- CalalignCondAPI ( Conditions API)
        +- FrontierDBReader (DBReader for Frontier)


CalalignDetectors (domain for Cal Align  detector specific stuff)
```

```
|
+- HCAL    (Hcal related DB-interface stuff)
+- HCALDB
+- ECALDB  (Source for ECAL DB-related stuff)
+- ECALDDL (DDL for ECAL)
+- Tracker
+- etc...
```

4.6 Bug Reporting and Change Control

For the CMS-Database project we can use the LCG Savannah setup (if possible) for the controlled change request and big fixing [need to discuss/decide with/by the CMS-DB project management]. However, the existing mechanism of Savannah does not support the automatic bug reporting to the developers.

4.7 Database Design Tools and Schema Review Process

In most cases, the schemas used for the various detector applications will be very similar and we would like to start with agreed upon patterns or templates. In fact, if we can stick to this approach we can share manpower to design and maintain the schemas for the majority of the needs. We propose Oracle designer be used for schema design and development work, including generating space reports and ddl's. It helps to provide consistent relationships among tables and it provides important documentation for the schemas in the form of ER diagrams. It is also a useful environment for sharing work among several developers. However, OD is not an easy tool to use for the beginner and many use Microsoft Visio or freeware tools to generate their schemas for prototyping purposes.

A review process will be established to go over schema and code that will go into production DB servers. This review will cover the following aspects:

- Compare the ER diagram with the DDLs.
- Examine DDL for tables, index, constraints, synonyms, grants, triggers.
- Assure Correct tablespace placement i.e. indexes in idx tablespaces.
- Every table, view and sequence must have a synonym.
- Check every object for storage definitions; match DDL with storage report, initial, next must be defined.
- Check all NOT NULL fields in a table are on top.
- Ensure that there are not multiple "Before Insert" triggers on the same table .
- Make sure the CLOBS and BLOBS go into their own tablespaces

For a more complete description of the review process used at Fermilab see [WWW]”DB checklist”.

We will establish oracle instances/servers to accommodate 1. Development, 2. Integration (final stage testing), and 3. Production. Existing servers have been installed at CERN and FNAL for this purpose and are:

```
*  list server specs and oracle instance names - uscmsdbdev

        Server name  Oracle Instance  location
        uscmsdb03  uscmscald  Fermilab Feynman Computing Center
        uscmsdb01  uscmscali,uscmscalp  Fermilab Feynman Computing Center
        bager  uscmscaldr Fermilab Wilson Hall
        uscmsdb02  cerncmscald?? CERN
```

5 Sub-system needs Descriptions

5.1 Overview

5.2 Possible Outline for Sub-system Documents

- A brief description of your sub-system.
- A description of how you plan to use the database (use cases or user narrative) for the cosmic challenge, detector commissioning, and physics running.
- A summary of what data will be stored, number of channels, size (bytes), the anticipated frequency of change. Also, anticipated access patterns for each type of data. This includes geometry and alignment, hardware configuration, electronics run configuration, calibration, and monitoring. Summarizing this information in tables will be very useful.
- A hierarchical overview of the database needs, and any schema designs you may already have.
- Your anticipated schedule and milestones.
- References to supporting documents.

6 Online detector specific schema designs descriptions and motivation

Frank and Online reps

7 Detector geometry database and alignment description and use

Oliver

8 Calibration overview and scale

Gaicamo

9 Generalized DB Design and Prototype

(Lee, Yuyi, Gennadiy, Frank) Generalized DB design and prototype for offline (and some online sub-detectors).

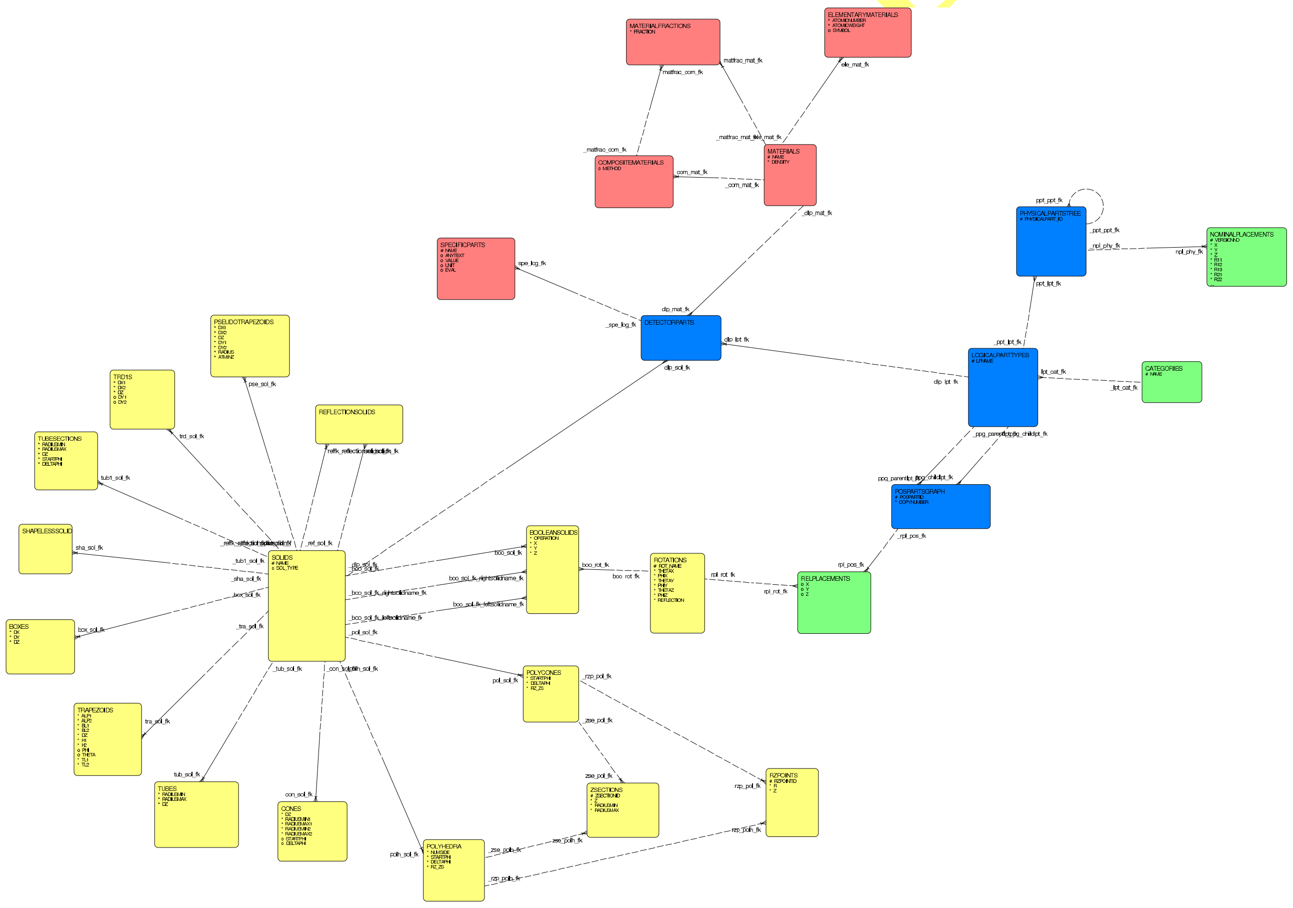

Figure 2: The Detector Geometry Database.

The CMS Detector Geometry Database (DGD) [2] schema design is shown in Figure 2. The DGD can be used as a starting point for schemas used to manage information about the detector construction, configuration, and data-taking conditions. The seeds of these extensions are described in the DGD document, and a CMS General purpose set of schemas have been built to explore this approach as shown in Figure 3. The DGD is the main entry into the system and it defines the detector hierarchy. Sub-detectors will reference the DGD and manage detector part relationships through it. Each database subsystem, indicated by a cylinder, will be deployed into a separate schema and tables containing history may also be optionally deployed into separate schemas. Such a generalized approach could meet the needs for all sub-detectors by using the super table design approach of DGD to accommodate additional types of information.

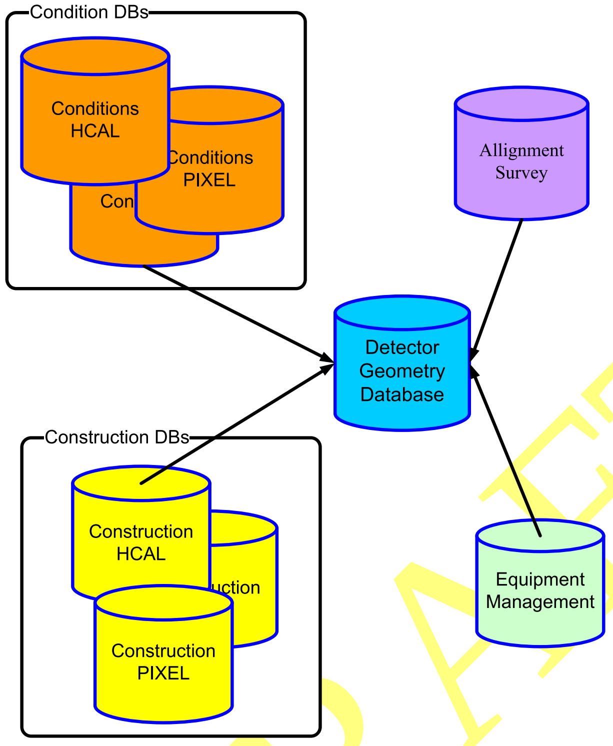

Figure 3: Adding Schemas to the Detector Geometry Database.

9.1 Extending the DGD Schema

The PhysicalPartsTree table, representing the expanded view of the DGD, is referenced by the construction and conditions database schemas. It contains instances of detector parts and can be extended to have other components, which are not included into the compact view of the DGD such as: hardware equipment in the counting room, cables, virtual components, and so on. Sub detector construction schemas are attached to the PPT and provide the details of the components used to build the complete detector, including construction testing, physical relationships, electrical signal connections, and additional attributes for each component. The conditions data is attached to the PPT through a dataset table that relates sets of conditions data to specific parts of the detector. Another schema is included to manage the Interval of Validity (IOV) for the conditions data.

This design offers a logical integration of existing database schemas (design blocks or subsystems) into one, while removing overlapping (redundant) schema components. It consists of four distinct blocks: hierarchal structure, signal connections, predefined attributes and history records. Some design-time constraints were replaced with run-time (logical or data driven) constraints, allowing one to establish relationships between detector parts at the super-type level. Attributes like positions, stages, grades, types or similar predefined attributes are selected into an attribute list and assigned to corresponding detector parts via their super-types as well. Each detector component type (kind of part) has a history table attached to it, which is populated by the corresponding database trigger, so component state history can be examined later if needed. An open design approach ensures the flexibility to accommodate additional schema blocks as needed.

Relationships between detector components, including configuration hierarchy and electronic signal connections, are maintained in the database and the histories tracked. For conditions data, validity intervals are recorded and mapped to sets of data through an Interval of Validity management portion of the schema. Tables are added to the component or data parts of the schemas as new kinds of parts, or data, are identified and tools are provided to facilitate this process. All portions of the schema together provide the complete functionality needed for CMS to maintain the information required by offline clients. The schema will also be used by some online detector sub-systems, specifically HCAL and PIXEL for which it was initially designed.

9.1.1 Construction/Configuration Schema

The design of the Construction/Equipment area uses a super table to which the many detector parts are connected (Figure 4 on the next page). It includes tables to record the relationships between the parts in the geometrical, mechanical, and electrical contexts. Constraints are maintained so only certain parts can be mounted on, or connected to other parts. Signal connections have similar constraints that only allow cables to be connected in a consistent fashion. Adding a new part is as simple as defining the attributes of the component, creating a table with these columns, and generating a relationship to the core tables. Tools are provided to easily add new parts as they are understood. Parts can be physical, like a module, or logical such as a part of a module. Parts can be detector pieces, electronics boards, crates, or anything having a description that can be put into a table.

9.1.2 Conditions Schema

The Conditions part of the schema (Figure 5 on page 19) also has a core table, and detector specific data tables that are attached to it. The core conditions table relates the data in the data tables to 1) the detector geometry and parts in the Construction/Equipment part of the schema, 2) Attributes of the data such as algorithm used to generate it, 3) the IOV management. The data is written into the data tables as sets, which are then

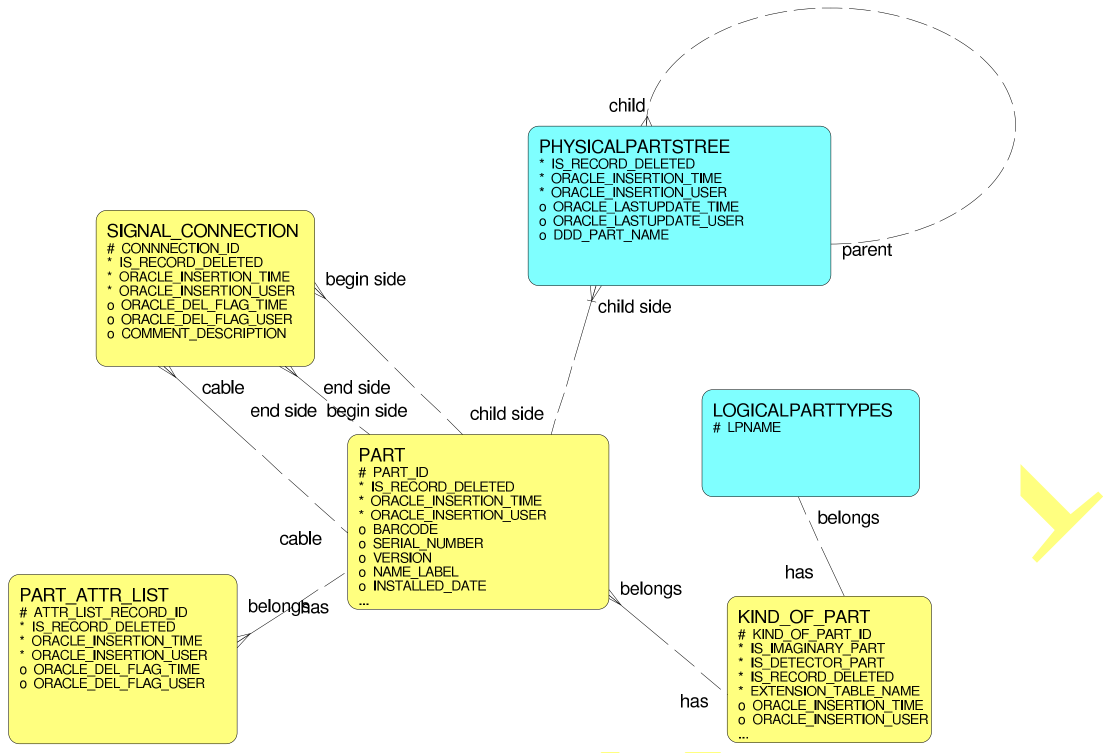

Figure 4: The Core Construction/Configuration Schema.

mapped to the proper Interval of Validity with a set of tools provided for this purpose. There are two categories of tables, 1) single version, and 2) multiple version. Single version tables are used to store monitoring data and have the IOV information (since or begin time and till or end time) recorded with each record in the table. Multiple version data tables are used for calibration and alignment data, have sets of channels that share a common IOV and other details of their origin, and can have more than one version. Adding additional calibration, alignment, or monitoring data is done by simply defining what is needed, and adding the appropriate table. Tools are provided to easily do this.

The content of each conditions table is entirely up to the person defining it. It can be related to the detector geometry and fully constrained, or it can have the channel index or mapping included in the table with no constraints. The data can be descrete values, integers, floats, strings, or even blobs depending on the specific need. Each “kind” of conditions table is uniquely identified by a specific table name.

9.1.3 Additional Attributes

Additonal attributes for the construction/configuration and conditions data sets are provided through a set of common attirbutes tables Figure 6 on page 20. Attributes include, for example, the name of hte algorithm used to create calibration or alignment values.

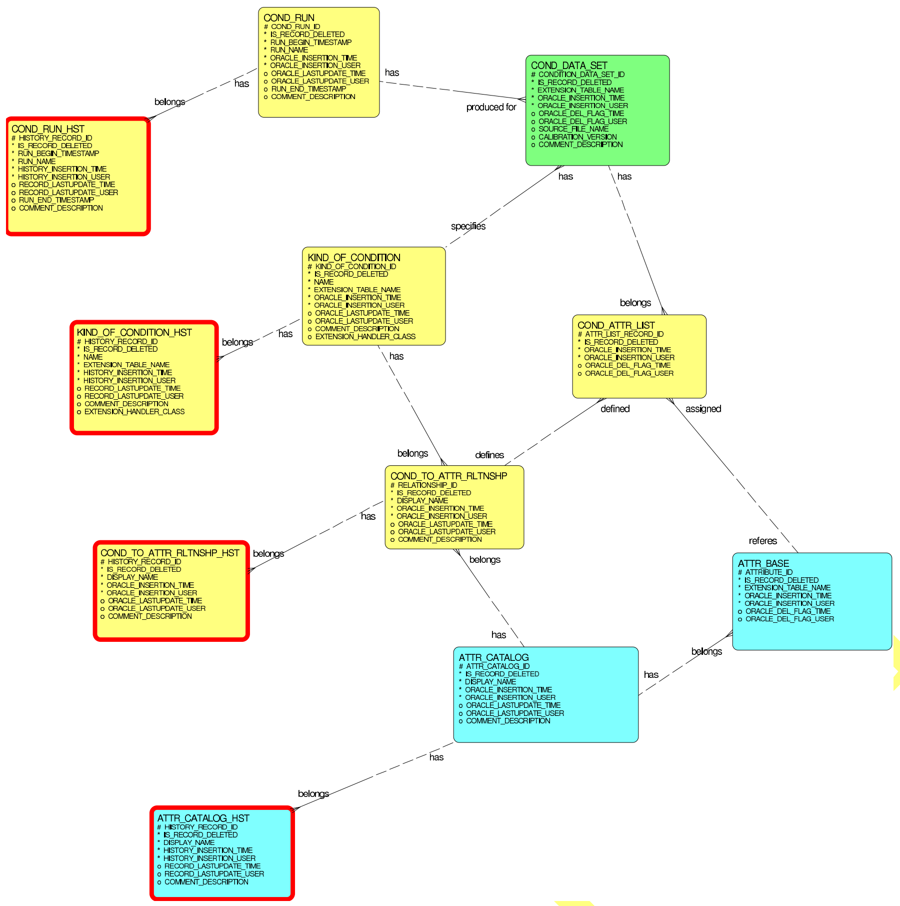

Figure 5: The Core Conditions Schema.

9.1.4 IOV Schema

The IOV management portion of the schema (Figure 7 on page 21) enables complex mapping between validity regions and the data in the tables. It also provides a mechanism for tagging certain data/algorithm/version combinations to easily allow offline access to a broad range of validity regions by specifying a single tag. The tag design is recursive, allowing the building of a tree structure so, for example, a single tag can refer to all appropriate calibration data for all detectors for times spanning many validity ranges.

10 Condition DB API and associated machinery.

(Based on report from API group)

(RALCondDB AttributeList vs Strongly Typed objects, DB access technologies incl. Frontier, odbc, etc.).

Coordinate with relevant parts of EDM and ORCA to use provided Cal/Align/etc.

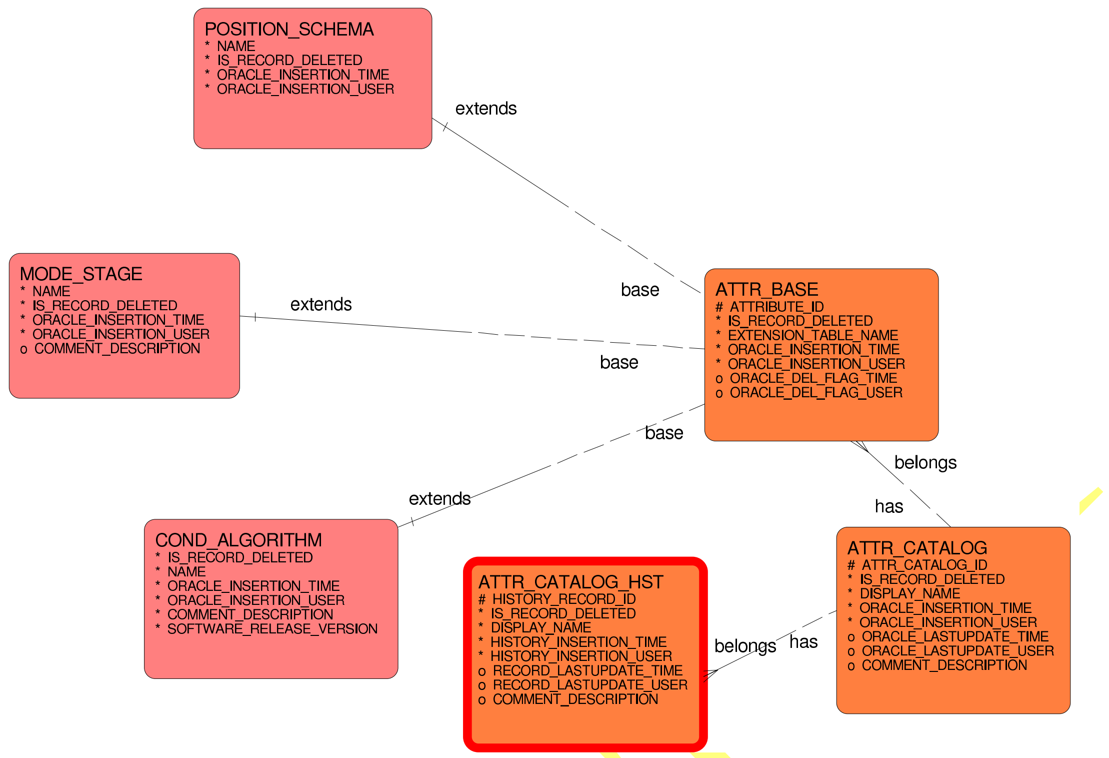

Figure 6: Attributes schema..

10.1 Overview

In April 2005 a group was convened to prepare a proposal for an extension to the Condition API that would enable the use of strongly typed objects.

The charge to this group was as follows:

1. Extend the existing condition API interface to include the ability to access strongly typed objects,
2. Insure that the software stack is compatible with bindings for various DB's already supported by RAL (OCI,ODBC,MySQL,SQLite, etc.), as well as Frontier.
3. Define the machinery needed to provide the extended functionality (e.g. code generation).
4. Understand how this machinery will work for schema evolution, and cases where existing database tables can be introduced to the system.
5. Establish a phased approach so we can proceed quickly with a solution, and implement more complex details (like compatibility with POOL) down the road. Ultimately, this may require interfacing with the POOL-RAL and COOL team to better understand how the changes we propose could be incorporated into their design.

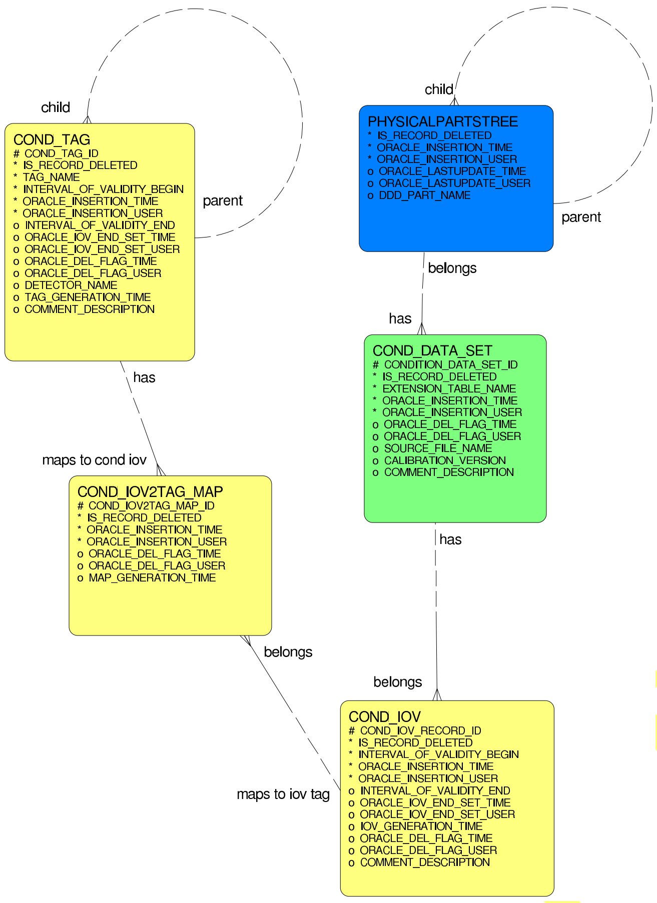

Figure 7: The Interval Of Validity Schema.

10.2 Prototypes

10.2.1 Prototype I: External Code-generation approach

Brief Description:

Prototype I was provided to demonstrate how database information can be mapped in to strongly typed objects by using code generation techniques to create C++ stubs and header files that are used for compilation. Figure 8 on the next page shows the application layers employed in this prototype. An EDM Context plug-in was created to provide the Framework access to the API. The DBReader is an abstract class that makes inserting DB access technologies straightforward. The concrete example that was produced is for accessing the HCAL testbeam database through the Frontier server, however it it would be possible to also use other DB API's such as OCI, ODBC, or POOL, to access the database. The details of this prototype are as follow:

1. The data object is described in an XML document as shown in this example:

```
<descriptor type="HCALGain" version="1" descriptorversion="1">
```

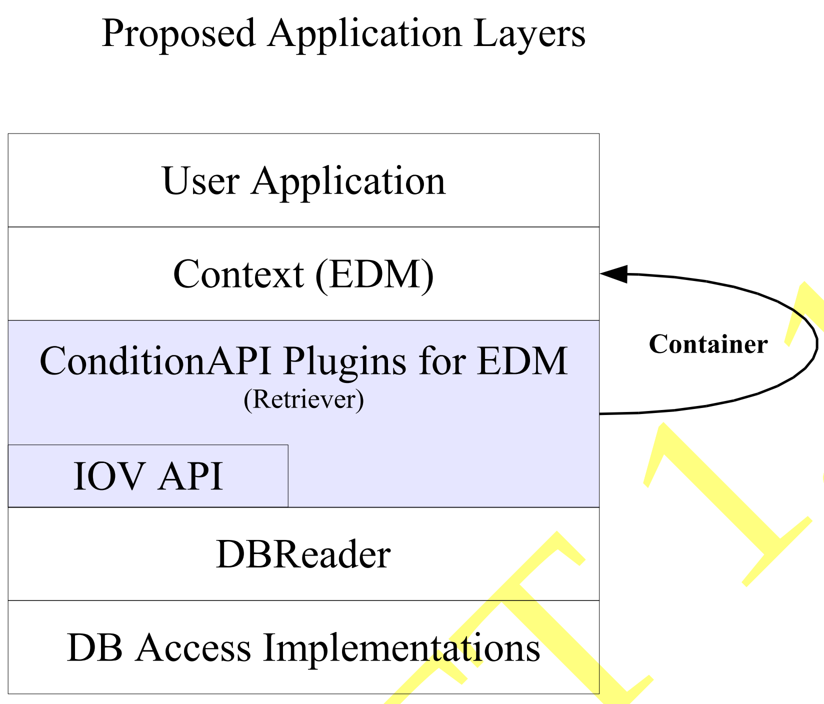
Proposed Application Layers
User Application
Context (EDM)
Container
ConditionAPI Plugins for EDM
(Retriever)
IOV API
DBReader
DB Access Implementations

Figure 8: Application layers for Proposal I.

```
  <attribute position="1"   type="int"    field="eta" />
  <attribute position="2"   type="double" field="phi" />
  <attribute position="3"   type="int"    field="depth" />
  <attribute position="4"   type="double" field="value" />
  <attribute position="5"   type="double" field="sigma" />
</descriptor>
```

2. A sub-detector can have many data objects, but they are all combined into single sub-detector plugin.

3. The code generator generated the data object C++ files and EDMContext plugin C++ files. Data object C++ files include: data object class, container class for data object, stub class which populates the container and returns IOV information. Figure 9 illustrates what is generated by the code generator. Users are free to

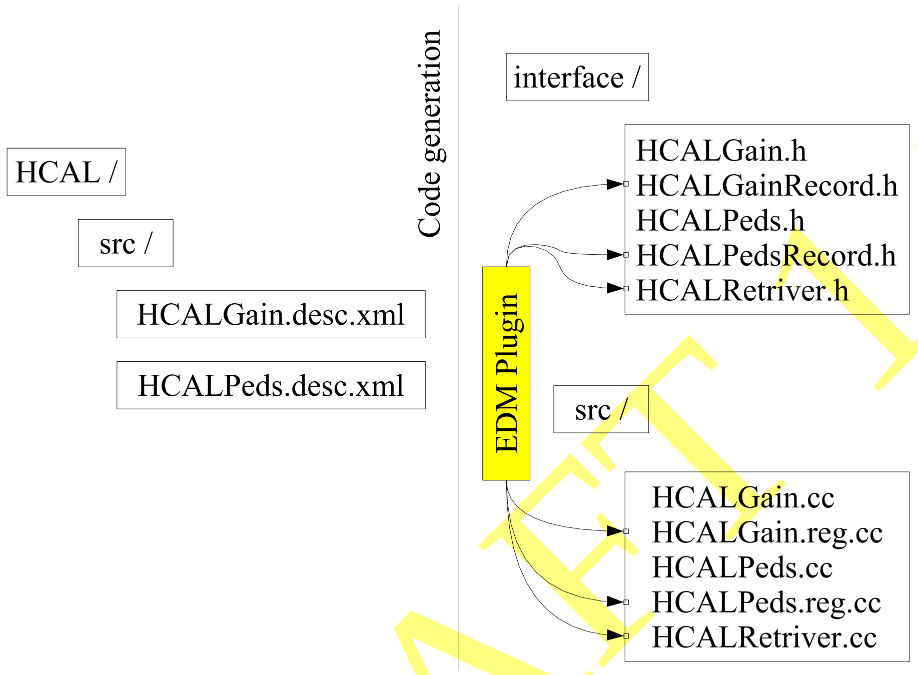

Figure 9: Code generation description.

specify their own containers for data objects. The only requirements for the user container is that it must have an "append(DataObject &dobj)" method. When users do not specify any container class, codegen creates a default container class for a data object, which is in essence, a sorted std::vector of data objects (it can be changed later).

4. The stub class uses the technology-independend DBReader interface (abstract C++ class) to send requests to a database and to get the result. DBReader provides the class method "getReader()" which return the particular implementation based on some logic (e.g. config files, environment variables, config server response, etc.)

The files generated from description above for HCAL Gain are included here as examples:

```
a) HCALGain.h
```

```
#ifndef _H_HCALGain_H
#define _H_HCALGain_H

#include "CalibTest/ConditionsAPI/interface/ConditionsAPI.h"

class HCALGain
 {
  public:
    int eta;
double phi;
int depth;
double value;
double sigma;


   // Other stuff
 };


class HCALGainContainer
 {
  private:
   std::vector<HCALGain> data;

  public:
   const std::vector<HCALGain>& getAll() const{return data;}
   const HCALGain& get(unsigned ind) const{return data[ind];} //XXX

   void append(DBReader *reader);
   void clear(){data.clear();}

   // Other stuff, e.g. iterators, cleaners, etc.
 };
```

b) Retriever (for both Gain and Peds)

```
class HCALRetriever : public edm::context::Producer,
                      public edm::context::ContextRecordIntervalFinder
{
   public:
      HCALRetriever(const std::string& iIOVToken);
      virtual ~HCALRetriever();

      const HCALGainContainer* produceHCALGain
        (const HCALGainRecord& );
      const HCALPedsContainer* produceHCALPeds
        (const HCALPedsRecord& );


   protected:
      //overriding from ContextRecordIntervalFinder
      virtual void setIntervalFor( const edm::context::ContextRecordKey&,
                                   const edm::Timestamp& ,
                                   edm::ValidityInterval& ) ;
   private:
      HCALRetriever(const HCALRetriever&); // stop default
      const HCALRetriever& operator=(const HCALRetriever& ); // stop default

      // ---------- member data --------------------------------

      HCALGainCond *cond_HCALGain_;
      HCALGainContainer data_HCALGain_;
      IOVType *iov_HCALGain_;


      HCALPedsCond *cond_HCALPeds_;
      HCALPedsContainer data_HCALPeds_;
      IOVType *iov_HCALPeds_;
}
```

10.2.2 Prototype II:POOL-ORA approach

Brief Description: The application software layers for prototype 2 are shown in Figure 10 on the next page illustrates what is generated by the code generator.

NOTE: This section is just a placeholder pending a more complete description from Vincenzo. This is from his initial email describing what he had in mind. I assume the actual prototype is different. LL

Based on the discussion we had yesterday this is my (Vincenzo's) proposal:

I will use as example the following payloads:

```
class Pedestals : public BaseCalib {
public:
         struct Item {
              float m_mean;
              float m_variance;
         };
         typedef long long ElectronicsId;
        typedef std::vector<Item>::const_iterator ItemIterator;
        typedef std::pair<ItemIterator,ItemIterator> ItemRange;
```

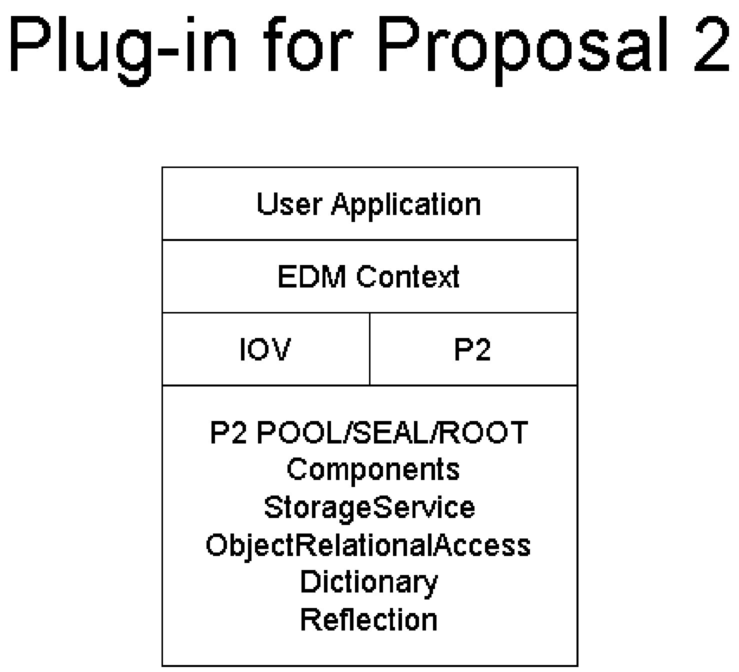

Figure 10: Software layers for Proposal II.

```
       ItermRange get(ElectronicsId id) const { return  ItemRange(m_pedestals[id].b

private:
          // the vector contains Item for a range of N electronic  channels staring
          // these channels are contigous also in terms of GeomIds...
    std::map<ElectronicsId,std::vector<Item> >  m_pedestals;
};

class Geom2Electronics : public BaseCalib {
     public:
         typedef long long GeomId;
         typedef long long ElectronicsId;
         typedef std::map<GeomId,ElectronicsId> Table;
         typedef Table::const_iterator TableIterator;
          ElectronicsId operator[](GeomId id) const {return  m_lookupTable[id];}
          const   std::map<GeomId,ElectronicsId> & lookupTable() const  { return
private:
       std::map<GeomId,ElectronicsId> m_lookupTable;
};

------------------------------------------------------------------ ---------
User Level:
                        context::Handle<Pedestals> pedestals;
                        context.get<EcalRecord>().get( pedestals);
                        context::Handle<Geom2Electronics>  lookUpTable;
                        context.get<EcalRecord>().get( lookUpTable);
                        for (Geom2Electronics::TableIterator  p=(*lookUpTable).look
                            Pedestals::ItemRange range =  (*pedestals)[p.second];
                            Geom2Electronics::GeomId id= p.first;
                           for (Pedestals::ItemIterator i=range.first;  i=range.sec
                                   std::cout << ++id << " " << *i << ",  ";
                          std::cout << std::endl;
                       }
------------------------------------------------------------------ ---------
Inside Context
                once per Calibration Type:
                   MetaData md(....);
                   const IOV * iov =  iovService.get(md);
                each time a new calib is required (event outside  current time-of-v
                   pair<IOV::TimeRange, CID> cid = (*iov).get(t);
                   const Calib * calib = DBService.read<Calib>(cid);    // or const
```

where: MetaData is the equivalent of "provenance" data in the EDM and is built by Context using info mainly from JobConfiguration. IOV is a collection of Interval-Of-Validities: in c++ a suitable implementation would be a std::map¡Time,CID¿ where the Key is the end-of-validity-time of the corresponding value: iov.upper_bound(t).second returns the CID valid at time t. in a RDBMS will be a table suitably indexed... CID is a calibration ID in an external service: typedef std::string CID is a reasonable implementation although knowledge of the internal structure surely helps in optimizing its storage in the IOV itself. COOL can be used as an actual implementation of the IOVService. COOL implements its own model of Versioning and naming of iovs It could be therefore an overkill if we decide to manage "MetaData" outside the IOVService itself.

DBservice is an abstract interface to the underlying implementation. In case of POOL:

```
   CID == Pool::Token and
   template<typename Calib> const Calib * PoolDBService::read(const  CID& cid) {
      pool::Ref<Calib> ref(sv,cid);
      return &(*ref);
   }
  In reality this template method cannot be virtual, so we may be  forced to introd
   template<typename Calib> const Calib * DBService::read(const CID&  cid) {
      return dynamic_cast<const Calib*> innerRead(cid);
   }
  with
  const BaseCalib * PoolDBService::innerRead(const CID& cid) {
      pool::Ref<BaseCalib> ref(sv,cid);
      return &(*ref);
   }
```

and having Pedestals etc inheriting from it... (we may need this base type also to let Context manage its own store...)

The careful reader has by now discovered that ConditionAPI is gone... Due to the intimacy between iov, metadata and context I see no role for it at least in this use-case. Other use cases (calibration studies, etc) not using the Context may profit of a ConditionAPI class that would help in managing iov, metadata etc....

11 Loading tools and access framework.

(Lee , Frank, Gennadiy, Michael) Loading tools and access framework. Incl. Online to offline transfer mechanism.

12 Delivery for distributed computing

(Lee, Emilio, Rep from Tier 1, Tier 2 center - Ian Fisk) Delivery for distributed computing: HLT farm and Offline processing and analysis.

Having a scalable database delivery architecture is important, especially in distributed environments such as the CMS offline and HLT farms. A multi-tier delivery system with data caching provides the desired features –scalability, performance, reliability – and can offer additional benefits by decoupling the designs of the client code and database schemas. Frontier [3] is a system built for Fermilab Run II which uses standard internet technologies to construct the middle tiers of the system and will be the baseline for CMS delivery of Database information to offline and HLT clients.

12.1 Testing for the HLT Environment

Testing will be done on the existing HLT filter farm to verify that the throughput and reliability of the Frontier server and caching meet the requirements. The following testing strategy has been outlined by Sergey Kosyakov.

There are two types of test possible:

1. Squid server load test, and
2. Full Frontier (server+client+Squid) test.

It would be good to have two dedicated nodes for the test: one for Squid server, and one for Tomcat/Frontier server. To begin with, we can use a single dual-CPU node with enough memory size as well.

For both options a host to run Squid is required. It can be any modern server with Linux. Important are:

1. Network connection - the faster the better, in my installation I got 79MB/sec on GigaEthernet
2. At least two disk partitions - one for software itself (can be shared "home" or any other partition), and one dedicated entirely for the cache.
3. Disk subsystem for the cache partition - the faster the better. Later on we can try memory-based FS, but it depends on overall objects size, while disk based cache should always work. The disk subsystem does not need redundancy, so if it is a RAID then the best would be the RAID level 0.Filesystem - is not that important, while I saw performance increase with XFS filesystem (comparing to Ext2 and 3) for 10MB/sec. noatime and async can increase performance. Cache partition size - depends on overall objects size, for testing 20GB (or less) should sufficient.
4. RAM - the more the better (for Linux to improve IO), 2GB should be enough to begin testing. To get optimization work for memory-based filesystems it would be better to have more RAM.
5. CPU - any modern one. I made tests on Athlon64 3400+. Xeon 3GHz and up should be good.
6. OS - Linux 2.6 is faster than 2.4

The Tomcat/Frontier server does not need a dedicated partition (only a regular one to keep software), it does not depend on disk IO, the only thing - it would be good to have at least 1GB of RAM. Good network connection is also important. CPU - the same as for Squid.

To perform the full Frontier test we need access to an Oracle database, and we need to create at least two tables on this DB - one for Frontier descriptors (small), and one to emulate data. So if you want to test queries which return 512MiB, then we need to populate some 512MiB of fake data into the database.The local database at FNAL could be used, but transatlantic calls would not be very representative.

Beyond GNU C/C++ compiler (v3.3 is good, but other versions could be considered as well) and standard Linux utilities, no other software is required. For Tomcat/Frontier server Java 1.4 is required.

13 GUI Framework evaluation and tools.

14 Deployment

The baseline database deployment is shown in the Figure 11 on the following page below. In this implementation there are three database instances for the production system: 1) the online database which maintains all configuration and conditions data needed to run and monitor the real time detector operation, 2) the Offline database having all the configuration and conditions information needed to debug the detector and perform offline data processing and analysis, and 3) a database which contains configuration and conditions information needed for the High Level Trigger (HLT) farm which is operated in real time near the detector and used to filter data on its way to permanent storage. These three database instances are referred to, for now, as Online Master Data Storage (OMDS), Offline Reconstruction Conditions DB for Offline (ORCOF), and Offline Reconstruction Conditions DB for Online (ORCON), respectively.

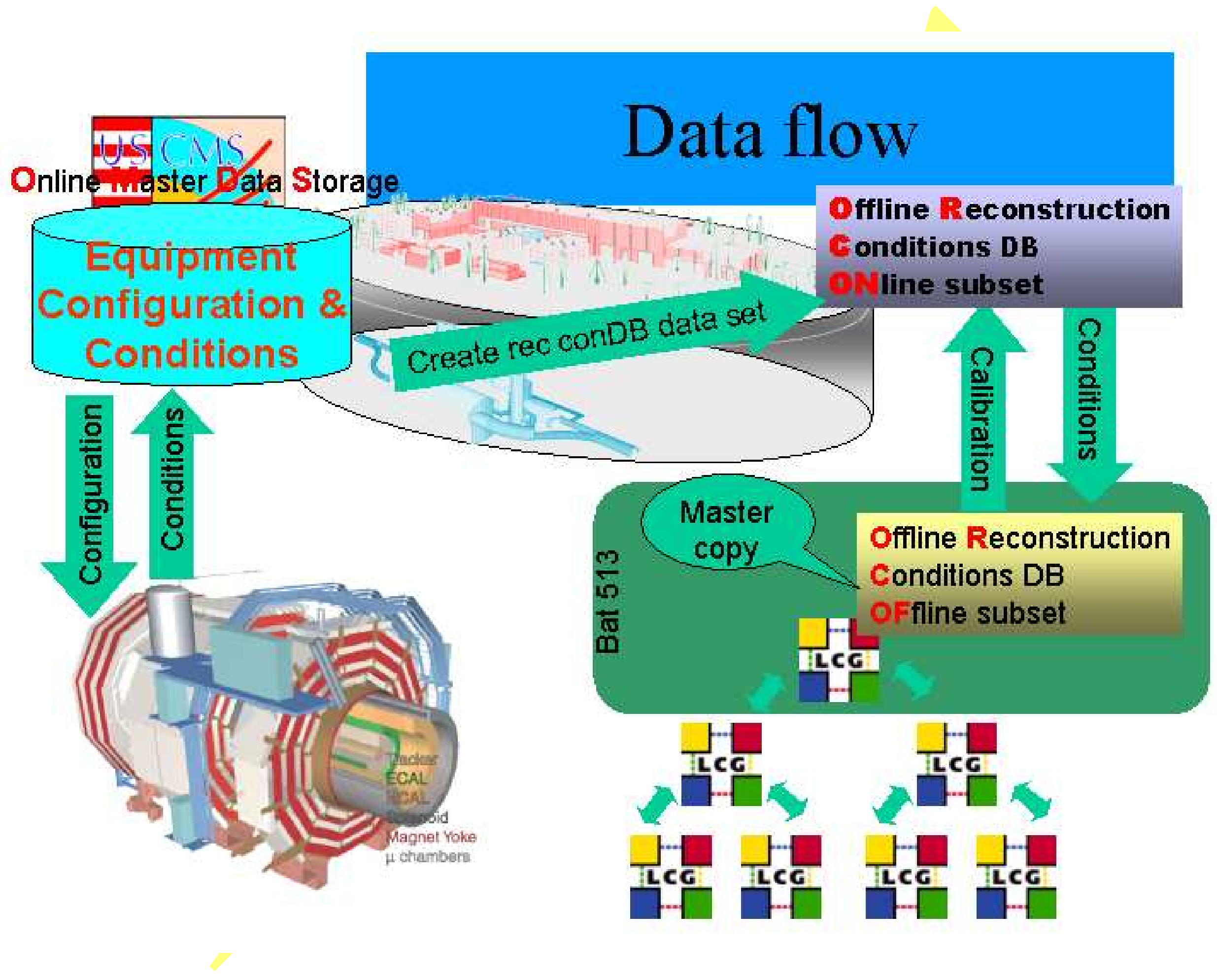
Data flow
Online Master Data Storage
Equipment Configuration & Conditions
Create rec conDB data set
Offline Reconstruction Conditions DB ONline subset
Calibration
Conditions
Configuration
Conditions
Master copy
Bat 513
Offline Reconstruction Conditions DB OFfline subset
LCG
Tracker
ECAL
HCAL
Solenoid
Magnet Yoke
μ chambers

Figure 11: Database deployment strategy.

A Glossary of Terms

It seems useful to agree up a set of terms to use for the various ideas we have been discussing. Here is a working list of the terms we have used. This list is an uneven mixture of items, some of which are very general and some of which are very specific.

Interval of Validity IOV - the time period for which a calibration , Slow Controls measurement, or configuration is valid. It has a beginning time called ”since time”, and an end time called ”till time”. The ending time can be undefined or infinite to indicate that the end of the validity region has not yet been determined.

Version - An identifier (usually made of numbers) which uniquely marks a calibration or configuration of a particular sub-detector for a particular IOV.

Tag - An identifier used to link sub-detector, version,IOV combinations together for ease of access.

Calibration Object (or Conditions Object) - The definition of the data content.

CalibrationObjectID - Specific instance of a Calibration Object, specified by type, IOV, and version or tag information.

sub-detector - Sub element of the complete CMS detector, for example ECAL, Muon Drift Tube, Tracker.

BLOB - Binary Large OBject, A database type for which the internal structure is not known by the database. Oracle allows very large blobs, up to many Gigabytes.

Bibliography

[1] A.T.M. Aerts, F. Glege,M. Liendl,I. Vorobiev,I.M. Willers,S. Wynhoff, **Status and Perspectives of Detector Databases in the CMS Experiment at the LHC**,29 October 2004 , CMS Note 2004-026.

[2] A.T.M. Aerts,M. Liendl,R. Gomez-Reino, **Detector Geometry Database**,April 2004 , CMS Note 2004-011.

[3] S. Kosyakov,et. al., **FroNtier: High Performance Database Access using Standard Web Components in a Scalable Multi-tier Architecture**,Presented at CHEP 2004, Interlaken Switzerland, September 2004. (Available at http://lynx.fnal.gov/ntier-wiki/Additional_20Documentation)